On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study

Fabio Palomba[*], Dario Di Nucci[*], Annibale Panichella[+], Rocco Oliveto[°], Andrea De Lucia[*]

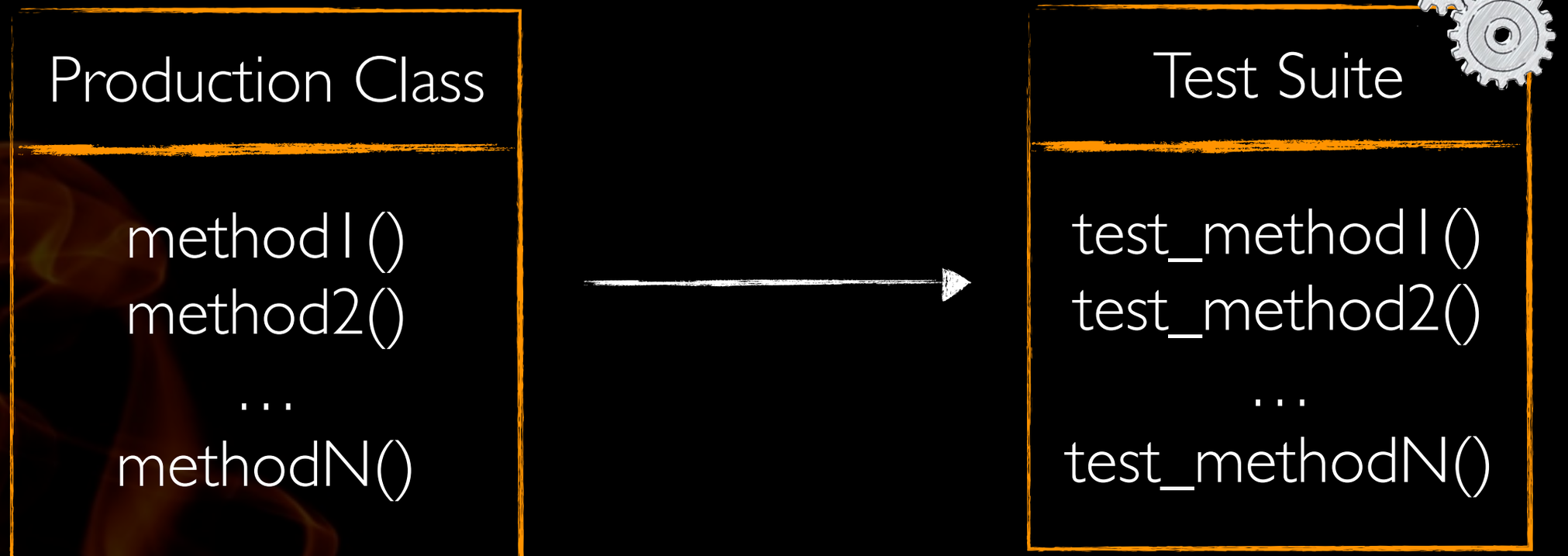
SBST 2016
May, 16th 2016
Austin, TX, USA

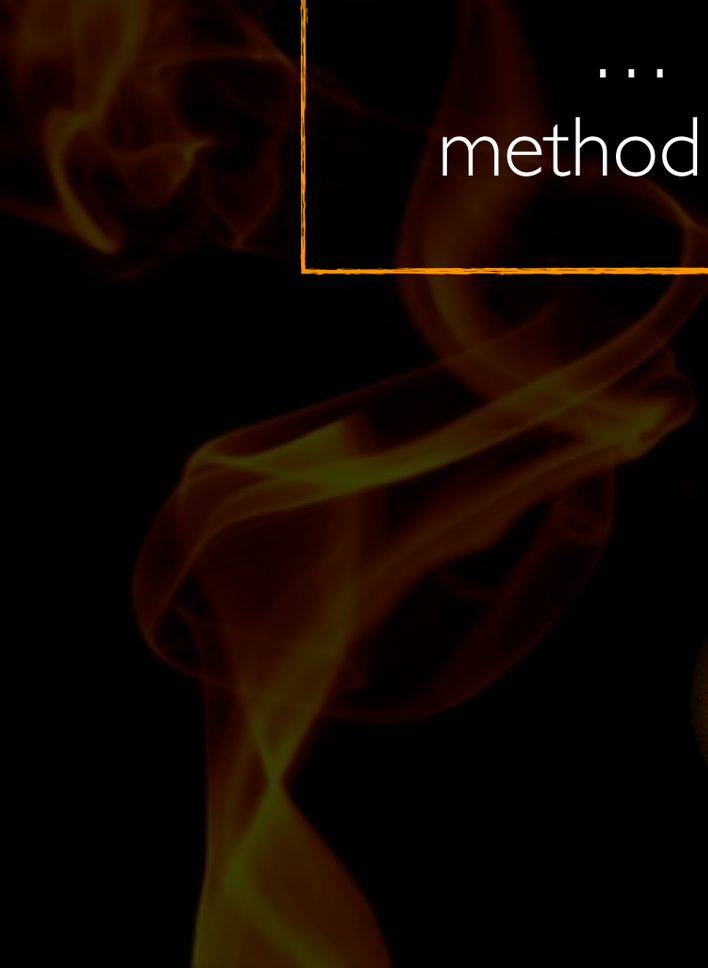
[*] University of Salerno, Italy
[+] Delft University of Technology, The Netherlands
[°] University of Molise, Italy

TU Delft

Delft
University of
Technology

Automatic Generation of Test Code





EVOSUITE

JTExpert

T3

Automatic Generation of Test Code

Effectiveness of Whole Suite Test Case Generation
[Arcuri and Fraser - SSBSE'14]

On The Effectiveness of
Whole Test Suite Generation

Andrea Arcuri[1] and Gordon Fraser[2]

[1] Simula Research Laboratory, P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no,
[2] University of Sheffield, Dep. of Computer Science, Sheffield, UK
gordon.fraser@sheffield.ac.uk

Abstract. A common application of search-based software testing is to generate test cases for all goals defined by a coverage criterion (e.g., statements, branches, mutants). Rather than generating one test case at a time for each of these goals individually, *whole test suite generation* optimizes entire test suites towards satisfying all goals at the same time. There is evidence that the overall coverage achieved with this approach is superior to that of targeting individual coverage goals. Nevertheless, there remains some uncertainty on whether the whole test suite approach might be inferior to a more focused search in the case of particularly difficult coverage goals. In this paper, we perform an in-depth analysis to study if this is the case. An empirical study on 100 Java classes reveals that indeed there are some testing goals that are easier to cover with the traditional approach. However, their number is not only very small in comparison with those which are only covered by the whole test suite approach, but also those coverage goals appear in small classes for which both approaches already obtain high coverage.

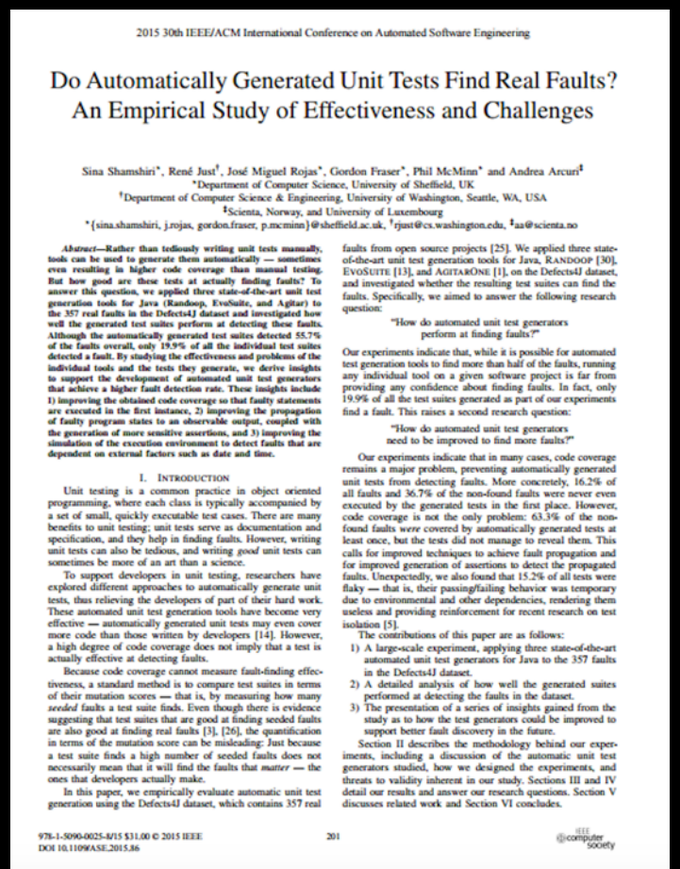
Key words: automated test generation, unit testing, search-based testing, EvoSuite

1 Introduction

Search-based software engineering has been applied to numerous different tasks in software development [15], and software testing is one of the most successful of these [1, 19]. One particular task in software testing for which search-based techniques are well suited is the task of automated generation of unit tests. For example, there are search-based tools like AUSTIN for C programs [18] or EvoSuite for Java programs [8].

In search-based software testing, the testing problem is cast as a search problem. For example, a common scenario is to generate a set of test cases such that their code coverage is maximized. A code coverage criterion describes a set of typically structural aspects of the system under test (SUT) which should be exercised by a test suite, for example all statements or branches. Here, the search space would consist of all possible data inputs for the SUT. A search algorithm (e.g., a genetic algorithm) is then used to explore this search space to find the

Effectiveness of Generated Test Cases on Effectiveness
[Shamshiri et al. - ASE'15]



2015 30th IEEE/ACM International Conference on Automated Software Engineering

Do Automatically Generated Unit Tests Find Real Faults?
An Empirical Study of Effectiveness and Challenges

Sina Shamshiri*, René Just†, José Miguel Rojas*, Gordon Fraser*, Phil McMinn* and Andrea Arcuri‡
*Department of Computer Science, University of Sheffield, UK
†Department of Computer Science & Engineering, University of Washington, Seattle, WA, USA
‡Scienta, Norway, and University of Luxembourg
*{sina.shamshiri, j.rojas, gordon.fraser, p.mcminn}@sheffield.ac.uk, †rjust@cs.washington.edu, ‡aa@scienta.no

***Abstract*—Rather than tediously writing unit tests manually, tools can be used to generate them automatically — sometimes even resulting in higher code coverage than manual testing. But how good are these tests at actually finding faults? To answer this question, we applied three state-of-the-art unit test generation tools for Java (Randoop, EvoSuite, and Agitar) to the 357 real faults in the Defects4J dataset and investigated how well the generated test suites perform at detecting these faults. Although the automatically generated test suites detected 55.7% of the faults overall, only 19.9% of all the individual test suites detected a fault. By studying the effectiveness and problems of the individual tools and the tests they generate, we derive insights to support the development of automated unit test generators that achieve a higher fault detection rate. These insights include 1) improving the obtained code coverage so that faulty statements are executed in the first instance, 2) improving the propagation of faulty program states to an observable output, coupled with the generation of more sensitive assertions, and 3) improving the simulation of the execution environment to detect faults that are dependent on external factors such as date and time.**

I. Introduction

Unit testing is a common practice in object oriented programming, where each class is typically accompanied by a set of small, quickly executable test cases. There are many benefits to unit testing; unit tests serve as documentation and specification, and they help in finding faults. However, writing unit tests can also be tedious, and writing *good* unit tests can sometimes be more of an art than a science.

To support developers in unit testing, researchers have explored different approaches to automatically generate unit tests, thus relieving the developers of part of their hard work. These automated unit test generation tools have become very effective — automatically generated unit tests may even cover more code than those written by developers [14]. However, a high degree of code coverage does not imply that a test is actually effective at detecting faults.

Because code coverage cannot measure fault-finding effectiveness, a standard method is to compare test suites in terms of their mutation scores — that is, by measuring how many *seeded* faults a test suite finds. Even though there is evidence suggesting that test suites that are good at finding seeded faults are also good at finding real faults [3], [26], the quantification in terms of the mutation score can be misleading: Just because a test suite finds a high number of seeded faults does not necessarily mean that it will find the faults that *matter* — the ones that developers actually make.

In this paper, we empirically evaluate automatic unit test generation using the Defects4J dataset, which contains 357 real faults from open source projects [25]. We applied three state-of-the-art unit test generation tools for Java, Randoop [30], EvoSuite [13], and AgitarOne [1], on the Defects4J dataset, and investigated whether the resulting test suites can find the faults. Specifically, we aimed to answer the following research question:

"How do automated unit test generators perform at finding faults?"

Our experiments indicate that, while it is possible for automated test generation tools to find more than half of the faults, running any individual tool on a given software project is far from providing any confidence about finding faults. In fact, only 19.9% of all the test suites generated as part of our experiments find a fault. This raises a second research question:

"How do automated unit test generators need to be improved to find more faults?"

Our experiments indicate that in many cases, code coverage remains a major problem, preventing automatically generated unit tests from detecting faults. More concretely, 16.2% of all faults and 36.7% of the non-found faults were never even executed by the generated tests in the first place. However, code coverage is not the only problem: 63.3% of the non-found faults *were* covered by automatically generated tests at least once, but the tests did not manage to reveal them. This calls for improved techniques to achieve fault propagation and for improved generation of assertions to detect the propagated faults. Unexpectedly, we also found that 15.2% of all tests were flaky — that is, their passing/failing behavior was temporary due to environmental and other dependencies, rendering them useless and providing reinforcement for recent research on test isolation [5].

The contributions of this paper are as follows:

1) A large-scale experiment, applying three state-of-the-art automated unit test generators for Java to the 357 faults in the Defects4J dataset.
2) A detailed analysis of how well the generated suites performed at detecting the faults in the dataset.
3) The presentation of a series of insights gained from the study as to how the test generators could be improved to support better fault discovery in the future.

Section II describes the methodology behind our experiments, including a discussion of the automatic unit test generators studied, how we designed the experiments, and threats to validity inherent in our study. Sections III and IV detail our results and answer our research questions. Section V discusses related work and Section VI concludes.

978-1-5090-0025-8/15 $31.00 © 2015 IEEE
DOI 10.1109/ASE.2015.86

IEEE computer society

Impact of Test Case Summaries on Bug Fixing Performance
[Panichella et al. - ICSE'16]

The Impact of Test Case Summaries on Bug Fixing Performance: An Empirical Investigation

Sebastiano Panichella,[1] Annibale Panichella,[2] Moritz Beller,[2]
Andy Zaidman,[2] Harald C. Gall[1]
[1]University of Zurich, Switzerland
[2]Delft University of Technology, The Netherlands
panichella@ifi.uzh.ch {a.panichella,m.m.beller,a.e.zaidman}@tudelft.nl gall@ifi.uzh.ch

ABSTRACT

Automated test generation tools have been widely investigated with the goal of reducing the cost of testing activities. However, generated tests have been shown not to help developers in detecting and finding more bugs even though they reach higher structural coverage compared to manual testing. The main reason is that generated tests are difficult to understand and maintain. Our paper proposes an approach, coined TestDescriber, which automatically generates test case summaries of the portion of code exercised by each individual test, thereby improving understandability. We argue that this approach can complement the current techniques around automated unit test generation or search-based techniques designed to generate a possibly minimal set of test cases. In evaluating our approach we found that (1) developers find twice as many bugs, and (2) test case summaries significantly improve the comprehensibility of test cases, which is considered particularly useful by developers.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code Inspections and Walk-throughs, Testing Tools*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Documentation, Enhancement*

Keywords

Software testing, Test Case Summarization, Empirical Study

1. INTRODUCTION

Software testing is a key activity of software development and software quality assurance in particular. However, it is also expensive, with overall testing consuming as much as 50% of overall project effort [8, 36], and programmers spending a quarter of their work time on developer testing [6].

Several search-based techniques and tools [16, 21, 40] have been proposed to reduce the time developers need to spend on testing by automatically generating a (possibly minimal) set of test cases with respect to a specific test coverage criterion [11, 21, 25, 28, 41, 43, 51, 54]. These research efforts produced important results: automatic test case generation allows developers to (i) reduce the time and cost of the testing process [5, 11, 13, 54]; to (ii) achieve higher code coverage when compared to the coverage obtained through manual testing [10, 22, 41, 43, 51]; to (iii) find violations of automated oracles (e.g. undeclared exceptions) [16, 22, 35, 40].

Despite these undisputed advances, creating test cases manually is still prevalent in software development. This is partially due to the fact that professional developers perceive generated test cases as hard to understand and difficult to maintain [18, 44]. Indeed, a recent study [23, 24] reported that developers spend up to 50% of their time in understanding and analyzing the output of automatic tools. As a consequence, automatically generated tests *do not improve the ability of developers to detect faults* when compared to manual testing [12, 23, 24]. Recent research has challenged the assumption that structural coverage is the only goal to optimize [1, 56], showing that when systematically improving the readability of the code composing the generated tests, developers tend to prefer the improved tests and were able to perform maintenance tasks in less time (about 14%) and at the same level of accuracy [18]. However, there is no empirical evidence that such readability improvements produce tangible results in terms of the number of bugs actually found by developers.

This paper builds on the finding that readability of test cases is a key factor to optimize in the context of automated test generation. However, we conjecture that the quality of the code composing the generated test cases (e.g., input parameters, assertions, etc.) is not the only factor affecting their comprehensibility. For example, consider the unit test `test0`[1] in Figure 1, which was automatically generated for the target class `Option`[2]. From a bird's-eye view, the code of the test is pretty short and simple: it contains a constructor and two assertions calling `get` methods. However, it is difficult to tell, without reading the contents of the target class, (i) what is the behavior under test, (ii) whether the generated assertions are correct, (iii) which if-conditions are eventually traversed when executing the test (coverage). Thus, we need a solution that helps developers to quickly understand both tests and code covered.

Paper contribution. To handle this problem, our paper proposes an approach, coined TestDescriber, which is designed to automatically generate summaries of the portion of code exercised by each individual test case to pro-

This paper was first submitted to the International Conference on Software Engineering (ICSE) for peer review in August 2015. It got accepted at ICSE'16. This is a pre-print copys.

[1]The test case has been generated using Evosuite [21].
[2]The class *Option* has been extracted from the *apache commons* library

Automatic Generation of Test Code

Usability of Automatic Generation Tools in Practice

[Rojas et al. - ISSTA'15]

Automated Unit Test Generation during Software Development: A Controlled Experiment and Think-Aloud Observations

José Miguel Rojas[1] Gordon Fraser[1] Andrea Arcuri[2]
[1]Department of Computer Science, The University of Sheffield, United Kingdom
[2]Scienta, Norway and SnT Centre, University of Luxembourg, Luxembourg

ABSTRACT

Automated unit test generation tools can produce tests that are superior to manually written ones in terms of code coverage, but are these tests helpful to developers *while they are writing code*? A developer would first need to know when and how to apply such a tool, and would then need to understand the resulting tests in order to provide test oracles and to diagnose and fix any faults that the tests reveal. Considering all this, does automatically generating unit tests provide any benefit over simply writing unit tests manually?

We empirically investigated the effects of using an automated unit test generation tool (EvoSuite) during development. A controlled experiment with 41 students shows that using EvoSuite leads to an average branch coverage increase of +13%, and 36% less time is spent on testing compared to writing unit tests manually. However, there is no clear effect on the quality of the implementations, as it depends on how the test generation tool and the generated tests are used. In-depth analysis, using five think-aloud observations with professional programmers, confirms the necessity to increase the *usability* of automated unit test generation tools, to *integrate* them better during software development, and to *educate* software developers on how to best use those tools.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

Keywords. Automated unit test generation, controlled experiment, think-aloud observations, unit testing

1. INTRODUCTION

Modern automated test generation tools and techniques can efficiently generate program inputs that lead to execution of almost any desired program location. In the context of automated unit test generation for object oriented software there are tools that exercise code contracts [1, 2] or parameterised unit tests [3], try to exhibit undeclared exceptions [1, 4], or simply aim to achieve high coverage [5–9]. There are even commercial tools like Agitar One [10] and Parasoft JTest [11] that generate unit tests that capture the current behaviour for automated regression testing. In previous work [12], we have shown that *testers* can achieve higher code coverage when using automatically generated tests than when testing manually. However, most previous investigations considered unit test generation independently of the *developers*: Does the use of an automated unit test generation tool support software developers in *writing code and unit tests*? And if not, how do automated unit test generation tools need to be improved in order to become useful?

In order to provide a better understanding of the effects automatic unit test generation has on software developers, we empirically studied a scenario where developers are given the task of implementing a Java class and an accompanying JUnit test suite. We studied this scenario using two empirical methods: First, we performed a controlled empirical study using 41 human subjects, who were asked to complete two coding and testing tasks, one manually, and one assisted by the EvoSuite unit test generation tool [13]. Second, we conducted five think-aloud observations with professional programmers to understand in detail how developers interact with the testing tool during implementation and testing. Our experiments yielded the following key results:

Effectiveness: Generated unit tests tend to have higher code coverage, and they can support developers in improving the coverage of their manually written tests (on average +7% instruction coverage, +13% branch coverage, +7% mutation score), confirming previous results [12]. However, the influence of automated unit test generation on the quality of the implemented software depends on how a tool and its tests are used.

Efficiency: When using automated unit test generation, developers spend less time on unit testing (36% less time in our experiments). Although participants commented positively on this time saving, it comes at the price of higher uncertainty about the correctness of the implementation, and we observe that spending more time with the generated tests leads to better implementations.

Usability: Readability of automatically generated unit tests is a key aspect that needs to be optimised (63% of participants commented that readability is the most difficult aspect of generated tests). If generated unit tests do not represent understandable, realistic scenarios, they may even have detrimental effects on software.

Integration: Test generation tools typically optimise for code coverage, but this is not what developers do. We observed three different approaches to unit testing, each posing different requirements on how testing tools should interact with developers and existing tests.

Education: Simply providing a test generation tool to a developer will not magically lead to better software—developers need education and experience on when and how to use such tools, and we need to establish best practices. For example, we found a moderate correlation between the frequency of use of our test generation tool and the number of implementation errors committed by developers.

The main contributions of this paper are the controlled empirical study (Section 3) and the think-aloud observations (Section 4),

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

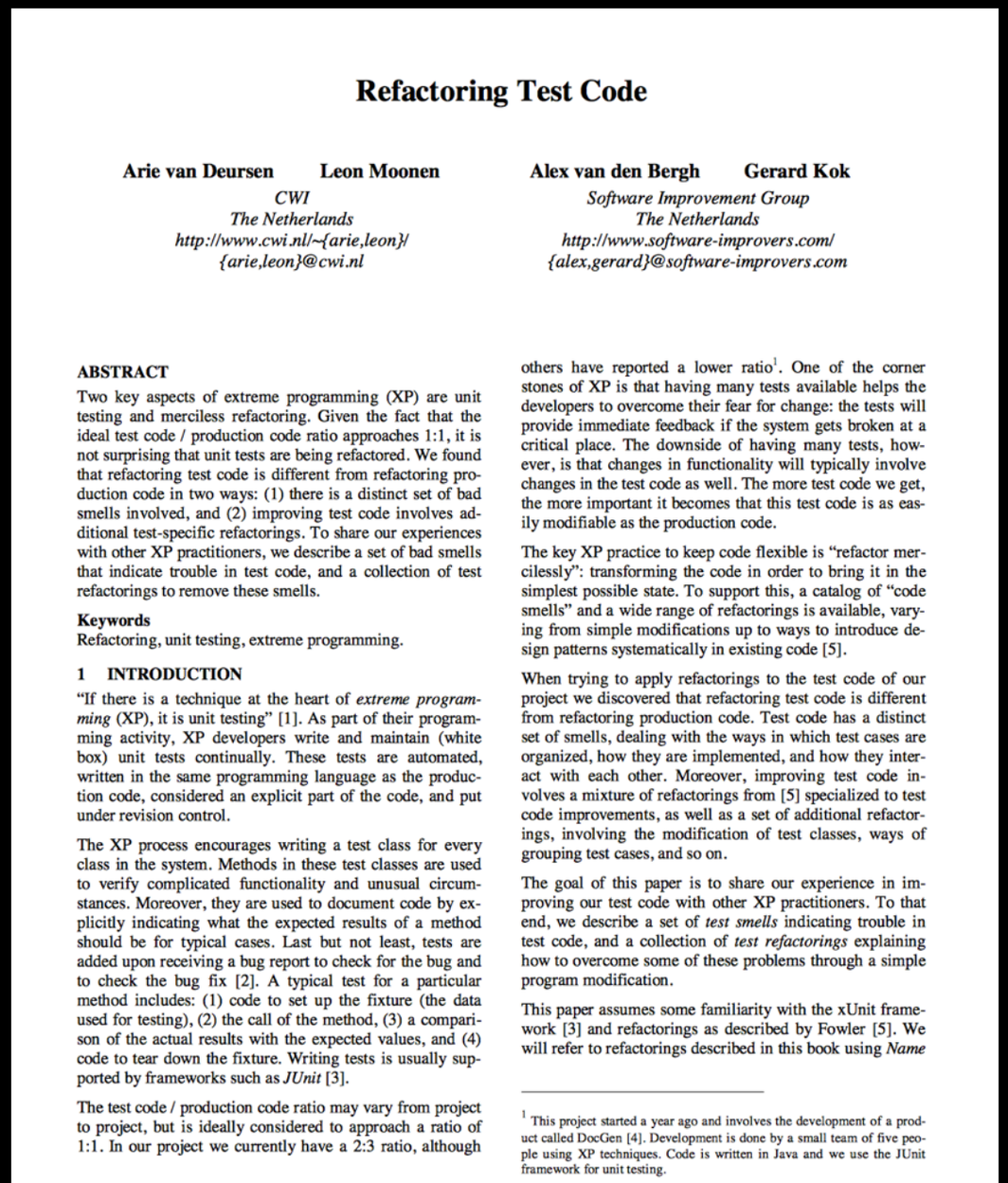
Copyright is held by the owner/author(s). Publication rights licensed to ACM.

ISSTA'15, July 13–17, 2015, Baltimore, MD, USA
ACM. 978-1-4503-3620-8/15/07
http://dx.doi.org/10.1145/2771783.2771801

Automatic Generation of Test Code

What about the characteristics of test code produced by such tools?

Test Smells in Test Code



Refactoring Test Code

Arie van Deursen **Leon Moonen**
CWI
The Netherlands
http://www.cwi.nl/~{arie,leon}/
{arie,leon}@cwi.nl

Alex van den Bergh **Gerard Kok**
Software Improvement Group
The Netherlands
http://www.software-improvers.com/
{alex,gerard}@software-improvers.com

ABSTRACT

Two key aspects of extreme programming (XP) are unit testing and merciless refactoring. Given the fact that the ideal test code / production code ratio approaches 1:1, it is not surprising that unit tests are being refactored. We found that refactoring test code is different from refactoring production code in two ways: (1) there is a distinct set of bad smells involved, and (2) improving test code involves additional test-specific refactorings. To share our experiences with other XP practitioners, we describe a set of bad smells that indicate trouble in test code, and a collection of test refactorings to remove these smells.

Keywords
Refactoring, unit testing, extreme programming.

1 INTRODUCTION

"If there is a technique at the heart of *extreme programming* (XP), it is unit testing" [1]. As part of their programming activity, XP developers write and maintain (white box) unit tests continually. These tests are automated, written in the same programming language as the production code, considered an explicit part of the code, and put under revision control.

The XP process encourages writing a test class for every class in the system. Methods in these test classes are used to verify complicated functionality and unusual circumstances. Moreover, they are used to document code by explicitly indicating what the expected results of a method should be for typical cases. Last but not least, tests are added upon receiving a bug report to check for the bug and to check the bug fix [2]. A typical test for a particular method includes: (1) code to set up the fixture (the data used for testing), (2) the call of the method, (3) a comparison of the actual results with the expected values, and (4) code to tear down the fixture. Writing tests is usually supported by frameworks such as *JUnit* [3].

The test code / production code ratio may vary from project to project, but is ideally considered to approach a ratio of 1:1. In our project we currently have a 2:3 ratio, although others have reported a lower ratio[1]. One of the corner stones of XP is that having many tests available helps the developers to overcome their fear for change: the tests will provide immediate feedback if the system gets broken at a critical place. The downside of having many tests, however, is that changes in functionality will typically involve changes in the test code as well. The more test code we get, the more important it becomes that this test code is as easily modifiable as the production code.

The key XP practice to keep code flexible is "refactor mercilessly": transforming the code in order to bring it in the simplest possible state. To support this, a catalog of "code smells" and a wide range of refactorings is available, varying from simple modifications up to ways to introduce design patterns systematically in existing code [5].

When trying to apply refactorings to the test code of our project we discovered that refactoring test code is different from refactoring production code. Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how they interact with each other. Moreover, improving test code involves a mixture of refactorings from [5] specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes, ways of grouping test cases, and so on.

The goal of this paper is to share our experience in improving our test code with other XP practitioners. To that end, we describe a set of *test smells* indicating trouble in test code, and a collection of *test refactorings* explaining how to overcome some of these problems through a simple program modification.

This paper assumes some familiarity with the xUnit framework [3] and refactorings as described by Fowler [5]. We will refer to refactorings described in this book using *Name*

[1] This project started a year ago and involves the development of a product called DocGen [4]. Development is done by a small team of five people using XP techniques. Code is written in Java and we use the JUnit framework for unit testing.

"Test Smells represent a set of a poor design solutions to write tests "

[Van Deursen et al. - XP 2001]

11

test smells related to the way developers write test fixtures and test cases

Test Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) "");
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

Test Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) "");
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

The test method checks the production method isGround()

Test Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) "");
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

But also the production method isVariable()

Test Smells in Test Code

```
public void test12 () throws Throwable {
JSTerm jSTerm0 = new JSTerm();
jSTerm0.makeVariable () ;
jSTerm0.add((Object) "");
jSTerm0.matches(jSTerm0);
assertEquals (false, jSTerm0.isGround ());
assertEquals(true, jSTerm0.isVariable());
}
```

This is an Eager Test, namely a test which checks more than one method of the class to be tested, making difficult the comprehension of the actual test target.

Test Smells in Test Code

A test case is affected by a Resource Optimism when it makes assumptions about the state or the existence of external resources, providing a non-deterministic result that depend on the state of the resources.

An Assertion Roulette comes from having a number of assertions in a test method that have no explanation. If an assertion fails, the identification of the assert that failed can be difficult.

Who cares about Test Smells?

Test Cases can be re-generated!

Who cares about Test Smells?
Test Cases can be re-generated!

True

Who cares about Test Smells?

Test Cases can be re-generated!

True

BUT

Who cares about Test Smells?
Test Cases can be re-generated!

True

BUT

Developers modify and remove test code

Developers add tests when automatic tools leave uncovered branches

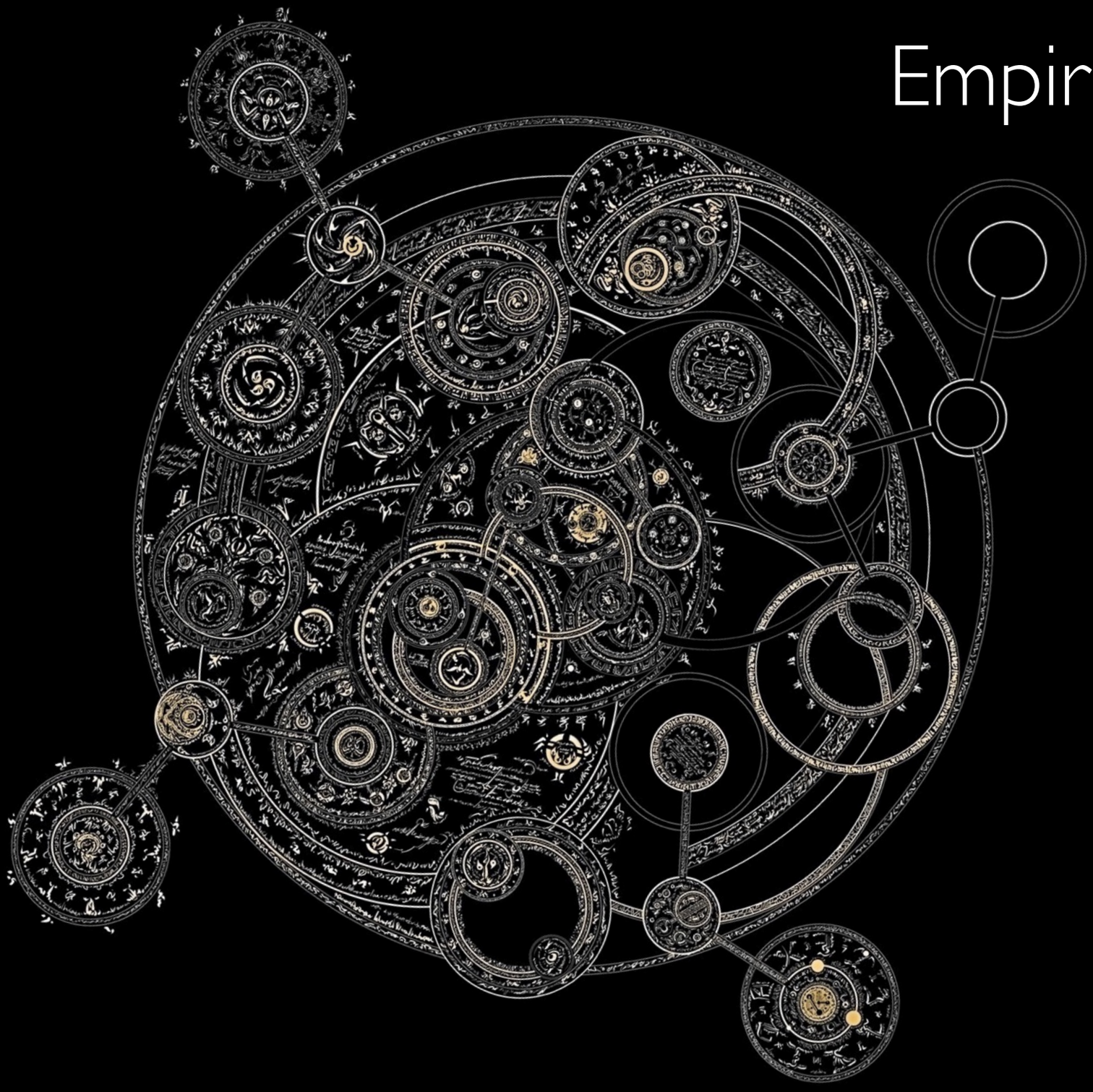
Developers combine generated with manually written tests

Usability of Automatic Generation Tools in Practice

[Rojas et al. - ISSTA'15]

On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study

Empirical Study Design



Empirical Study Design

8

test smell types

"Refactoring Test Code"

[Van Deursen et al. - XP 2001]

Empirical Study Design

8 test smell types

"Refactoring Test Code"
[Van Deursen et al. - XP 2001]

110 software projects

"A Large Scale Evaluation of Automated Unit Test Generation using Evosuite"
[Fraser and Arcuri - TOSEM 2014]

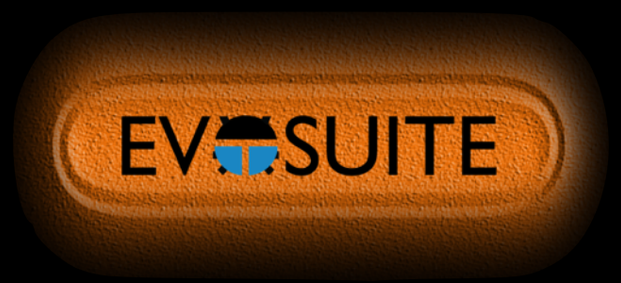
Empirical Study Design

8 test smell types

"Refactoring Test Code"
[Van Deursen et al. - XP 2001]

110 software projects

"A Large Scale Evaluation of Automated Unit Test Generation using Evosuite"
[Fraser and Arcuri - TOSEM 2014]



Empirical Study Design

16,603

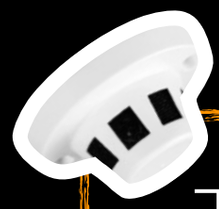
JUnit classes

"A Large Scale Evaluation of Automated Unit Test Generation using Evosuite"

[Fraser and Arcuri - TOSEM 2014]

EVOSUITE

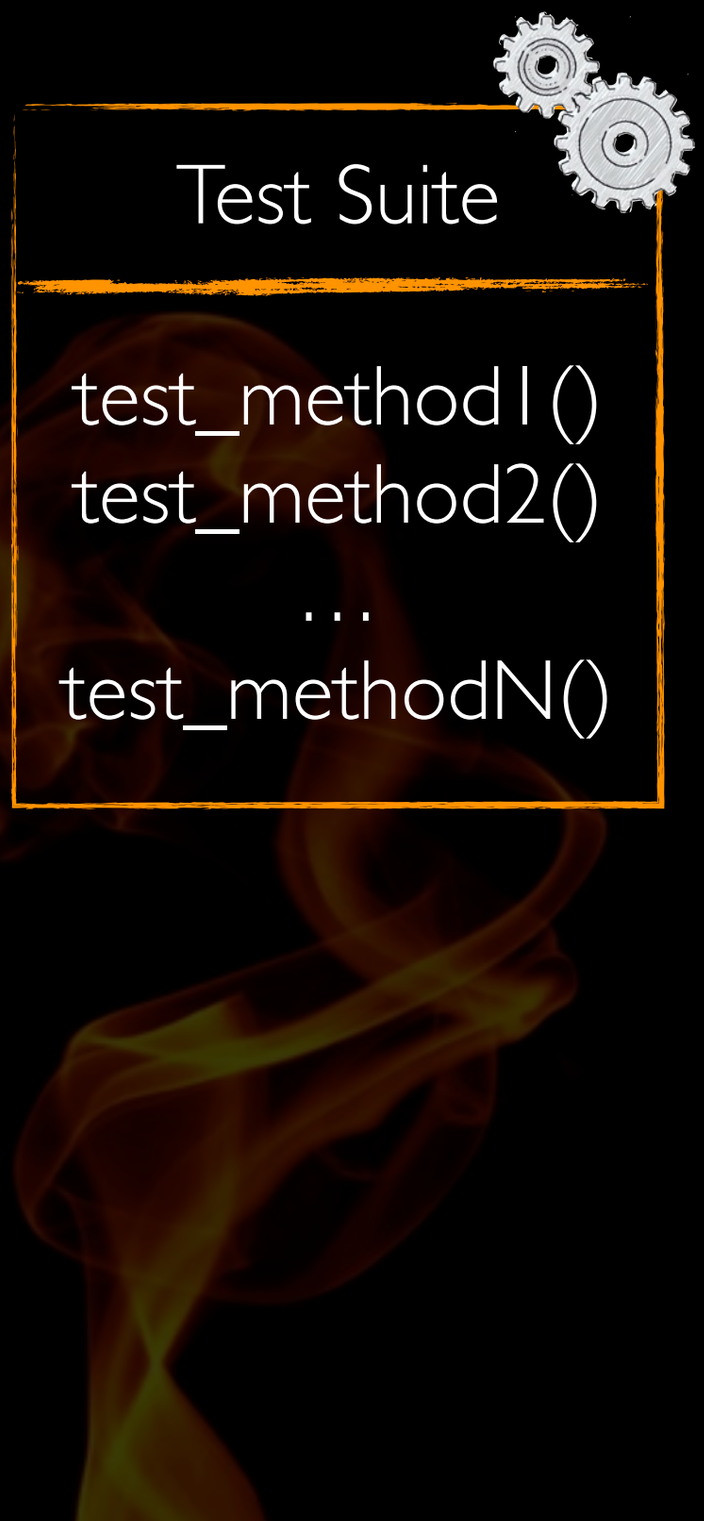
Data Extraction

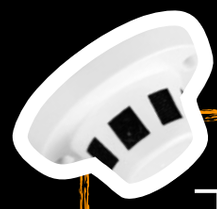


Test Smell Detector

"Are Test Smells Harmful? An Empirical Study"
[Bavota et al. - EMSE 2015]

Data Extraction





Test Smell Detector

"Are Test Smells Harmful? An Empirical Study"
[Bavota et al. - EMSE 2015]

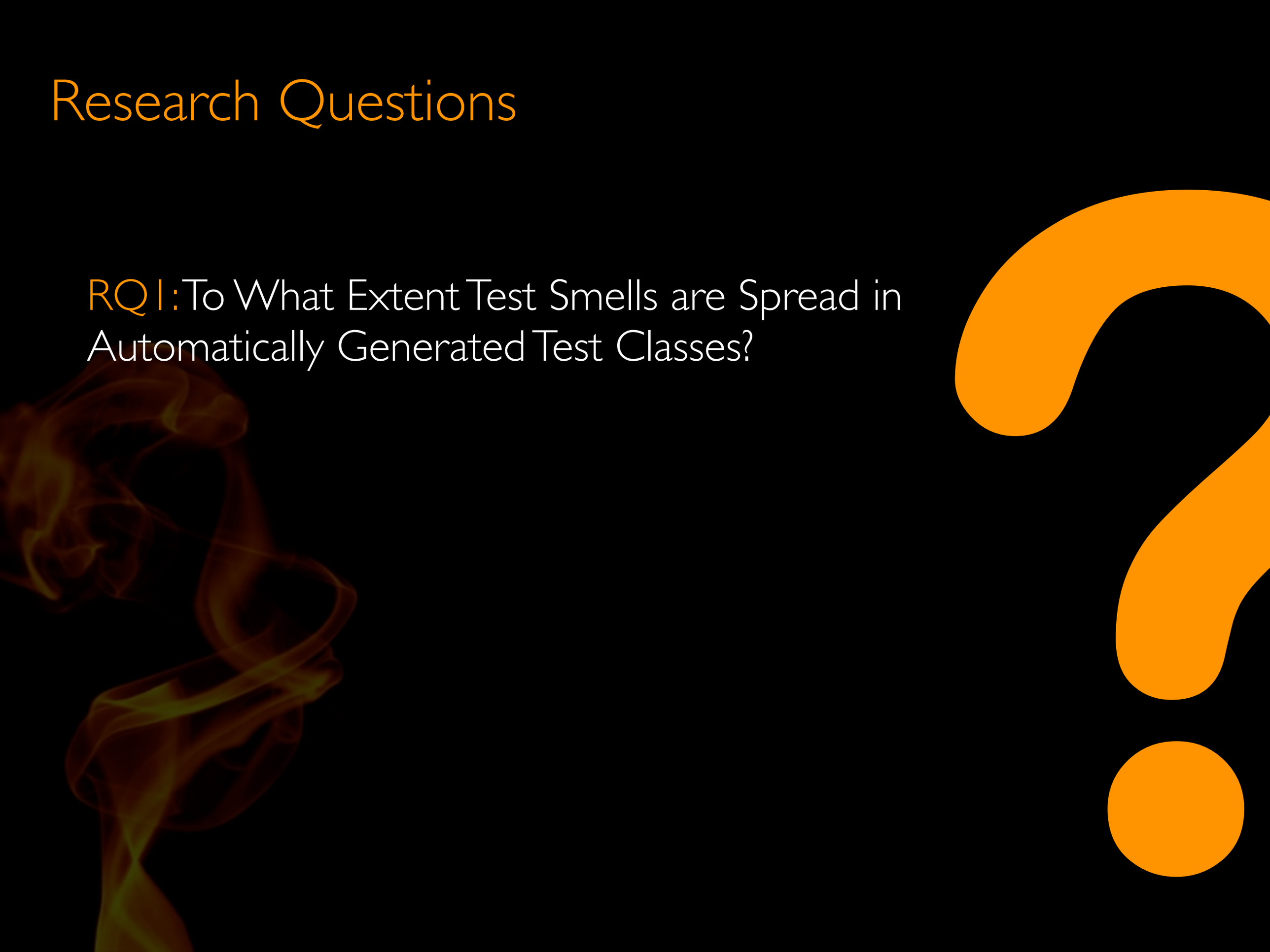
75% precision

100% recall

Sample size: 378 JUnit classes

Research Questions

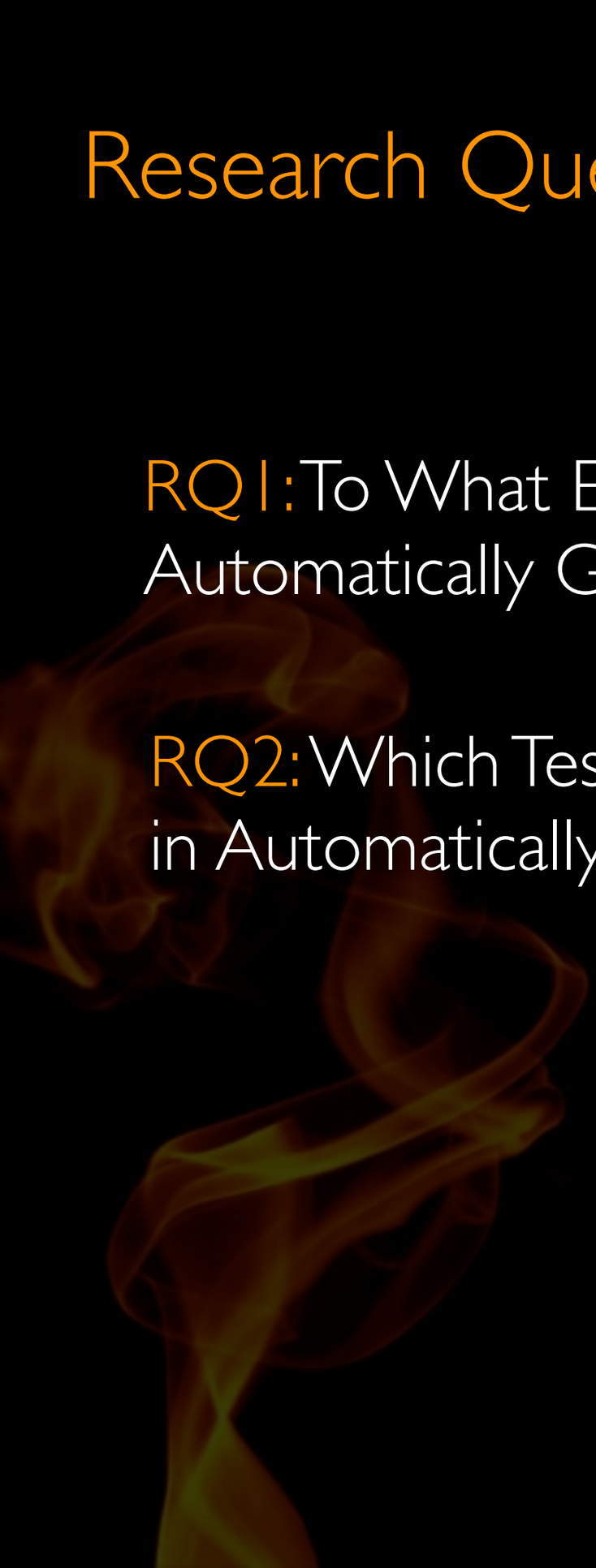
RQ 1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?



Research Questions

RQ1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

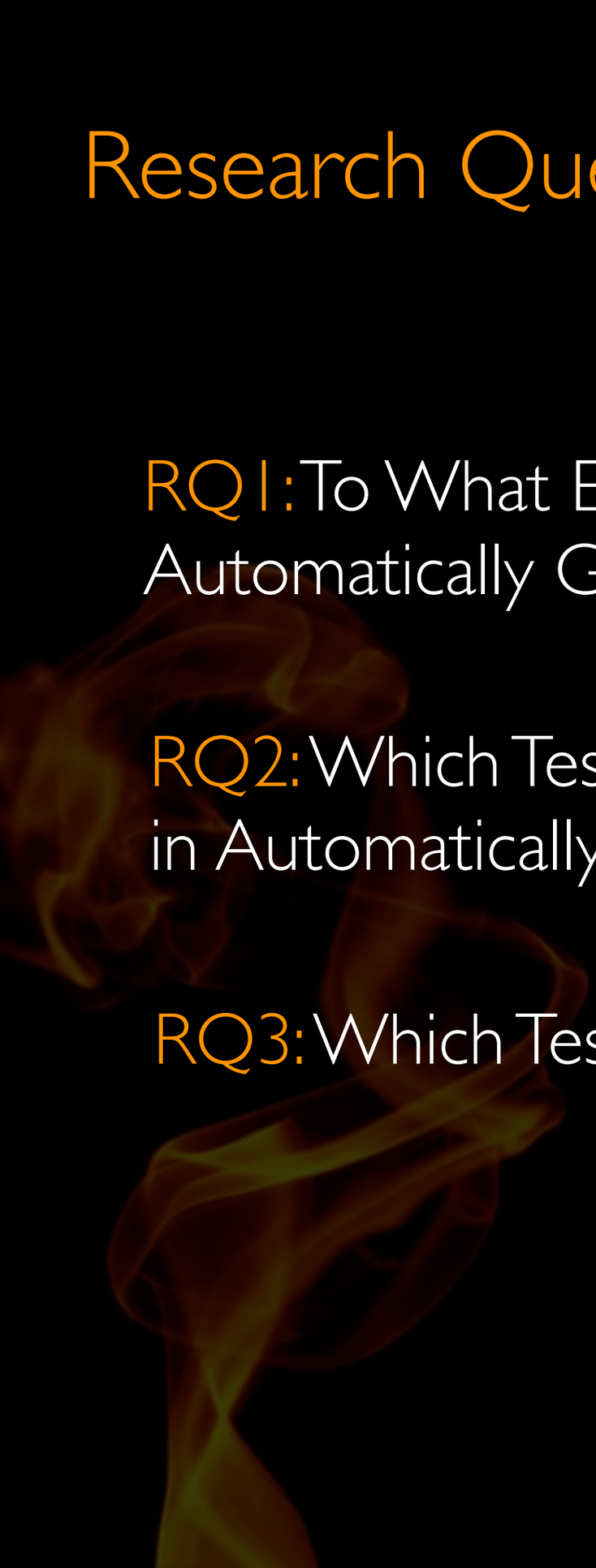


Research Questions

RQ1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

RQ3: Which Test Smells Co-Occur Together?



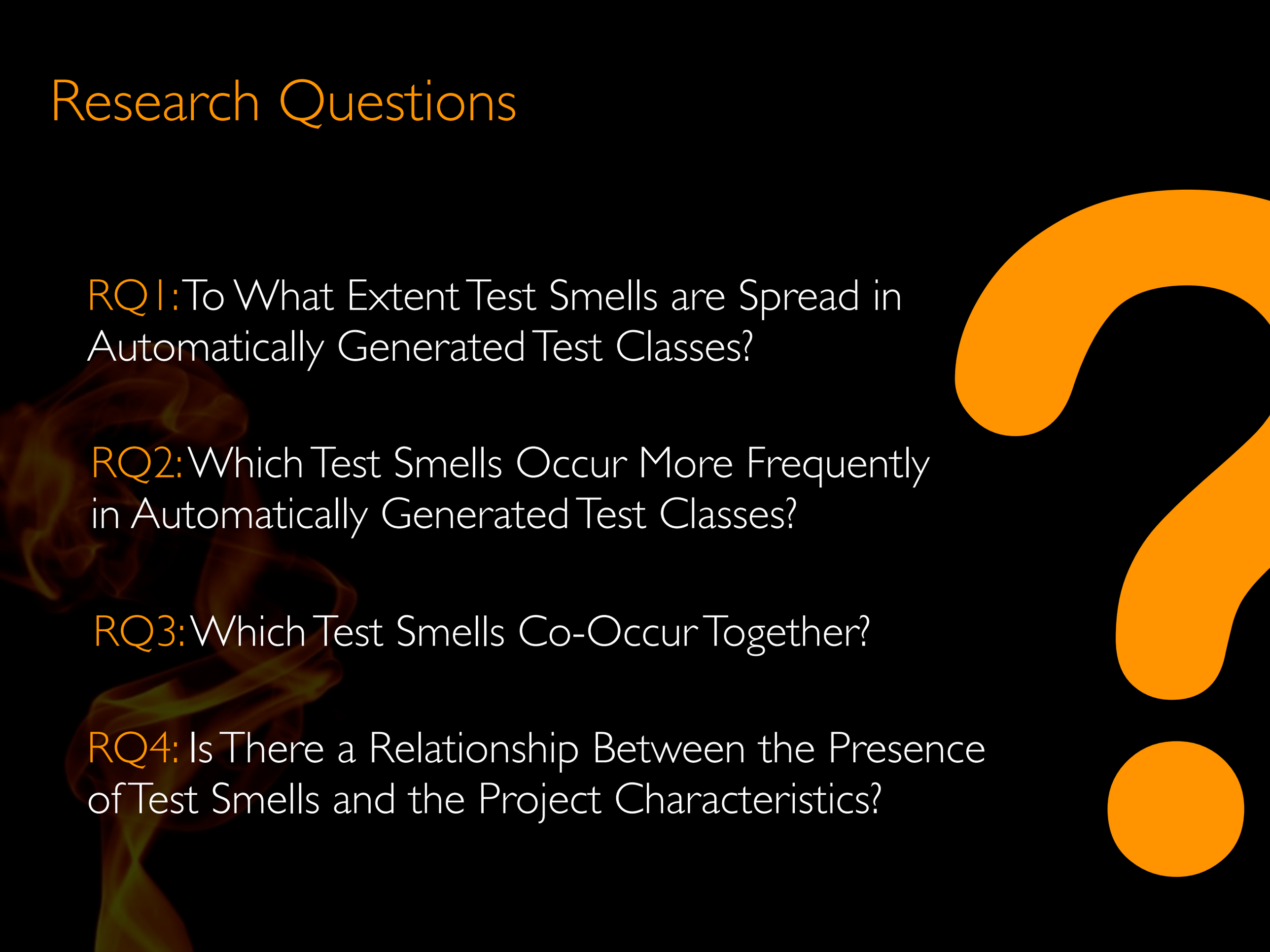
Research Questions

RQ1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

RQ3: Which Test Smells Co-Occur Together?

RQ4: Is There a Relationship Between the Presence of Test Smells and the Project Characteristics?



On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study

Analysis of the Results

Results of the Study

RQ1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?

13,791!

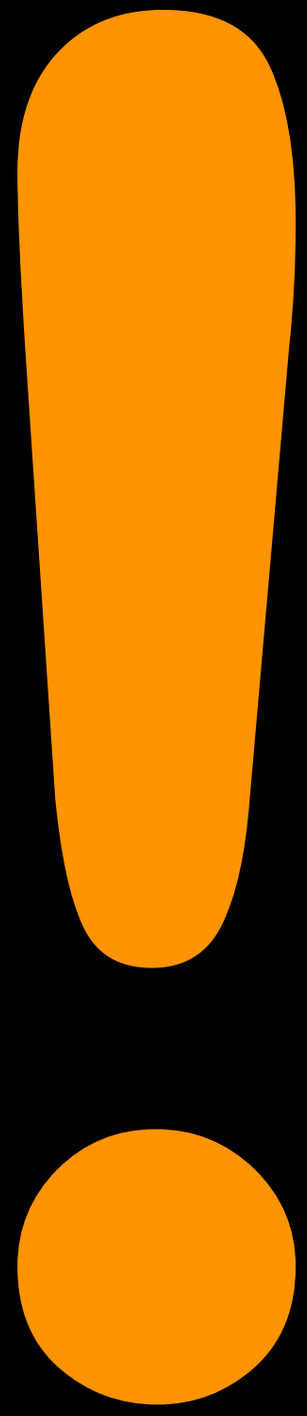
smelly JUnit classes

Results of the Study

RQ1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?

83%

of the JUnit classes analyzed

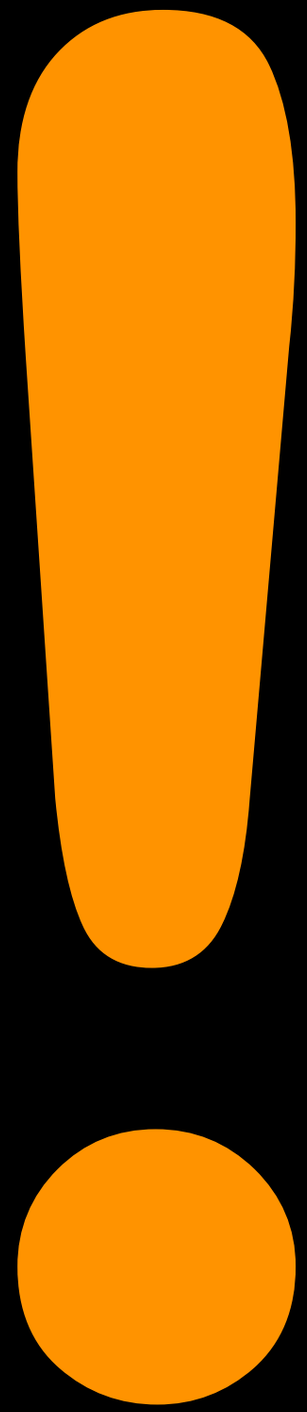


Results of the Study

RQ1: To What Extent Test Smells are Spread in Automatically Generated Test Classes?

RQ 1

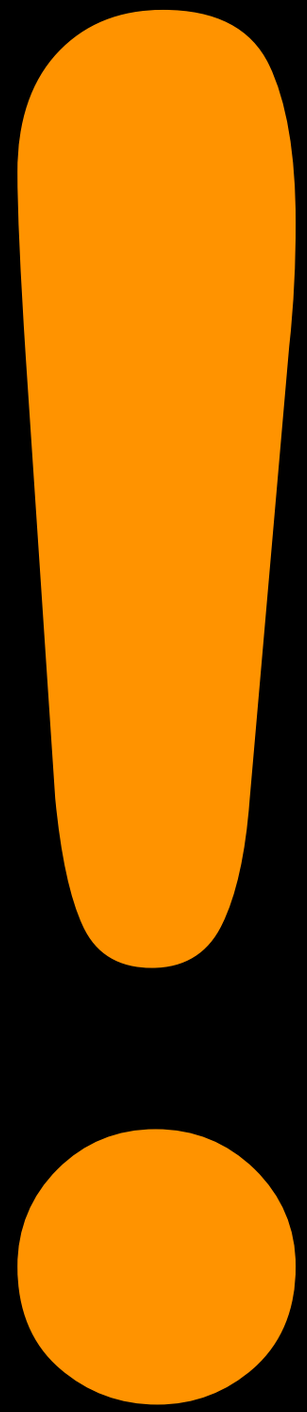
Test Smells are highly diffused in the automatically generated test suites



Results of the Study

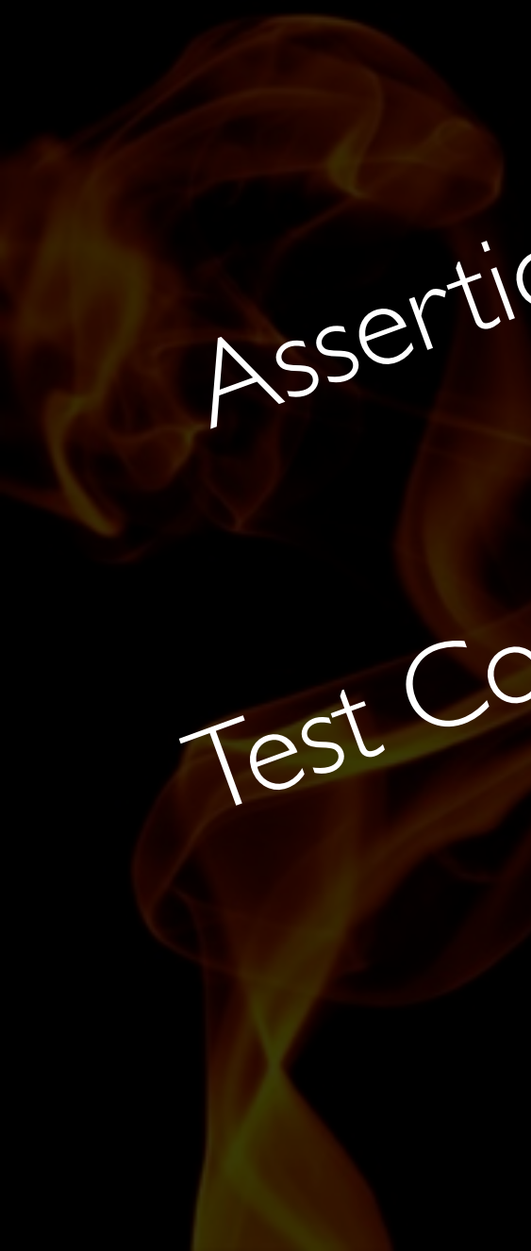
RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

54%



Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?



Assertion Roulette

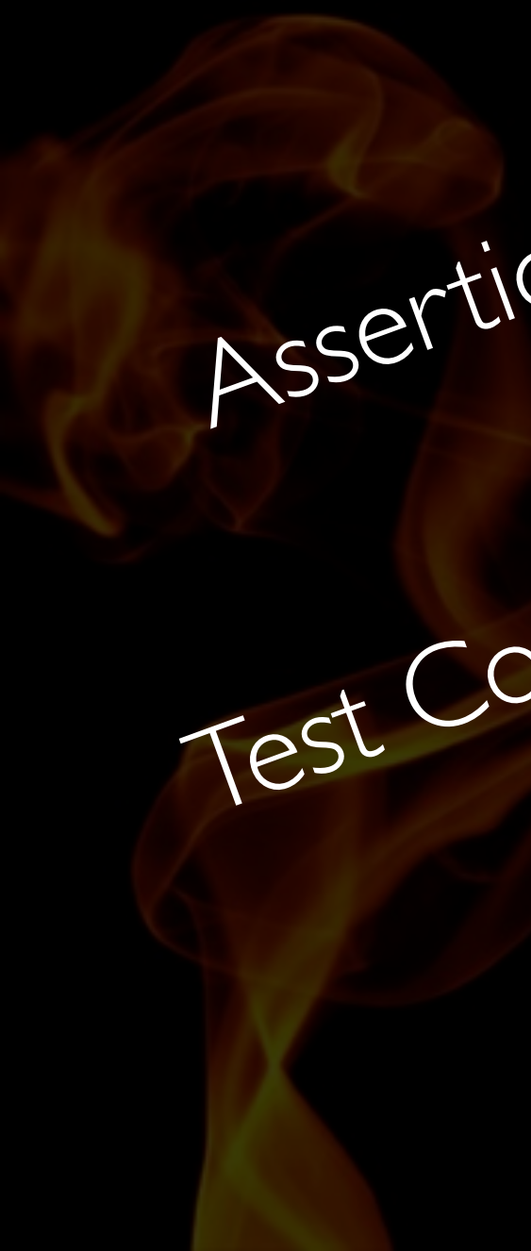
54%

Test Code Duplication

33%

Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?



Assertion Roulette 54%

Test Code Duplication 33%

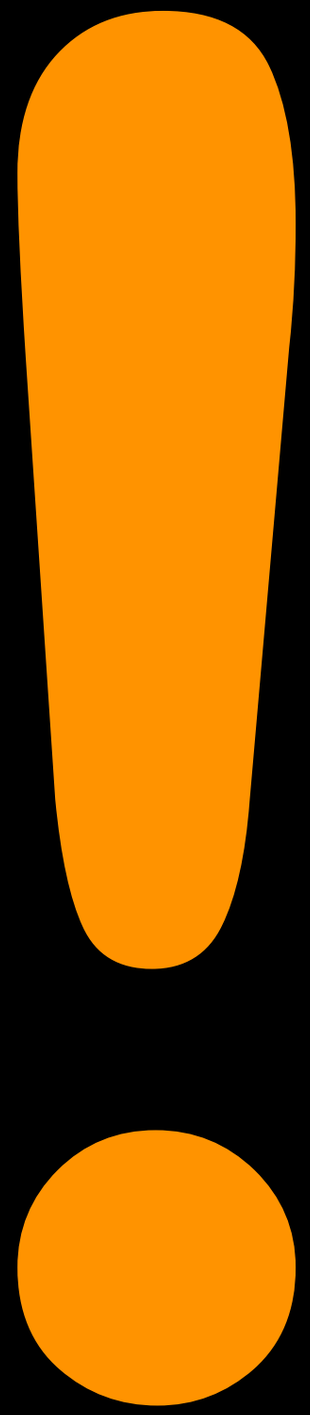
Eager Test 29%

Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

Assertion Roulette

```
public void test8 () throws Throwable {
Document document0 = new Document();
assertNotNull(document0);
document0.procText.add((Character) ”s”);
String string0 = document0.stringify();
assertEquals (“s”, document0.stringify());
assertNotNull(string0);
assertEquals(“s”, string0);
}
```



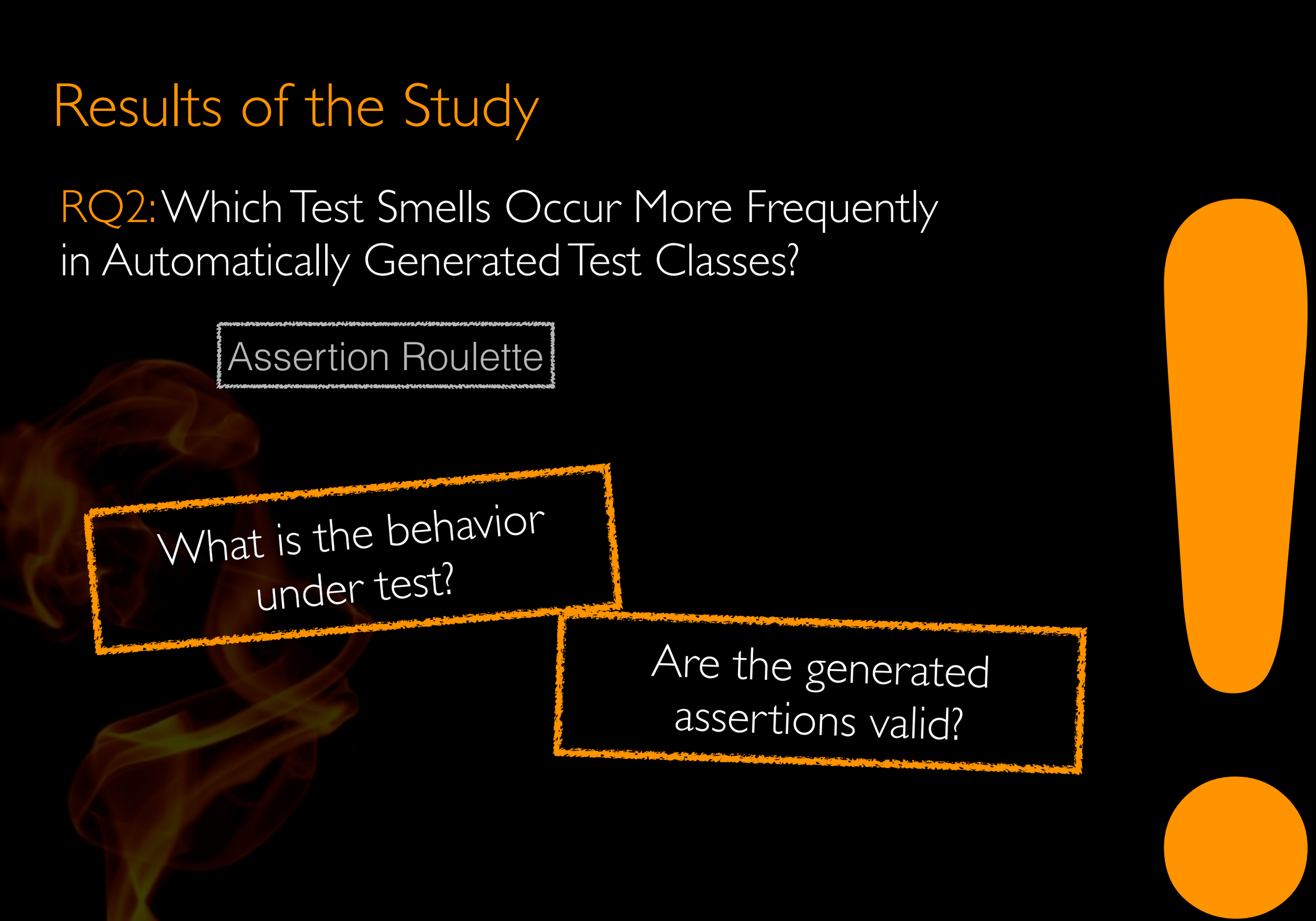
Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

Assertion Roulette

What is the behavior under test?

Are the generated assertions valid?



Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

Assertion Roulette

These problems have a huge impact on developers' ability to find faults

The Impact of Test Case Summaries on Bug Fixing Performance
An Empirical Investigation

[Panichella et al. - ICSE'16]

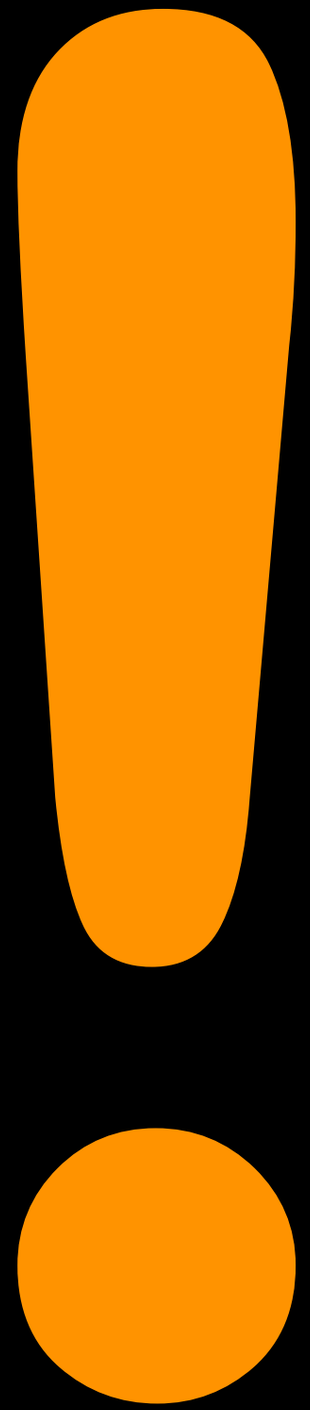
Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

Test Code Duplication

```
public void test8 () throws Throwable {
GenericProperties generic0 = new GenericProperties();
boolean boolean0 = generic0.isValidClassname();
…
}
```

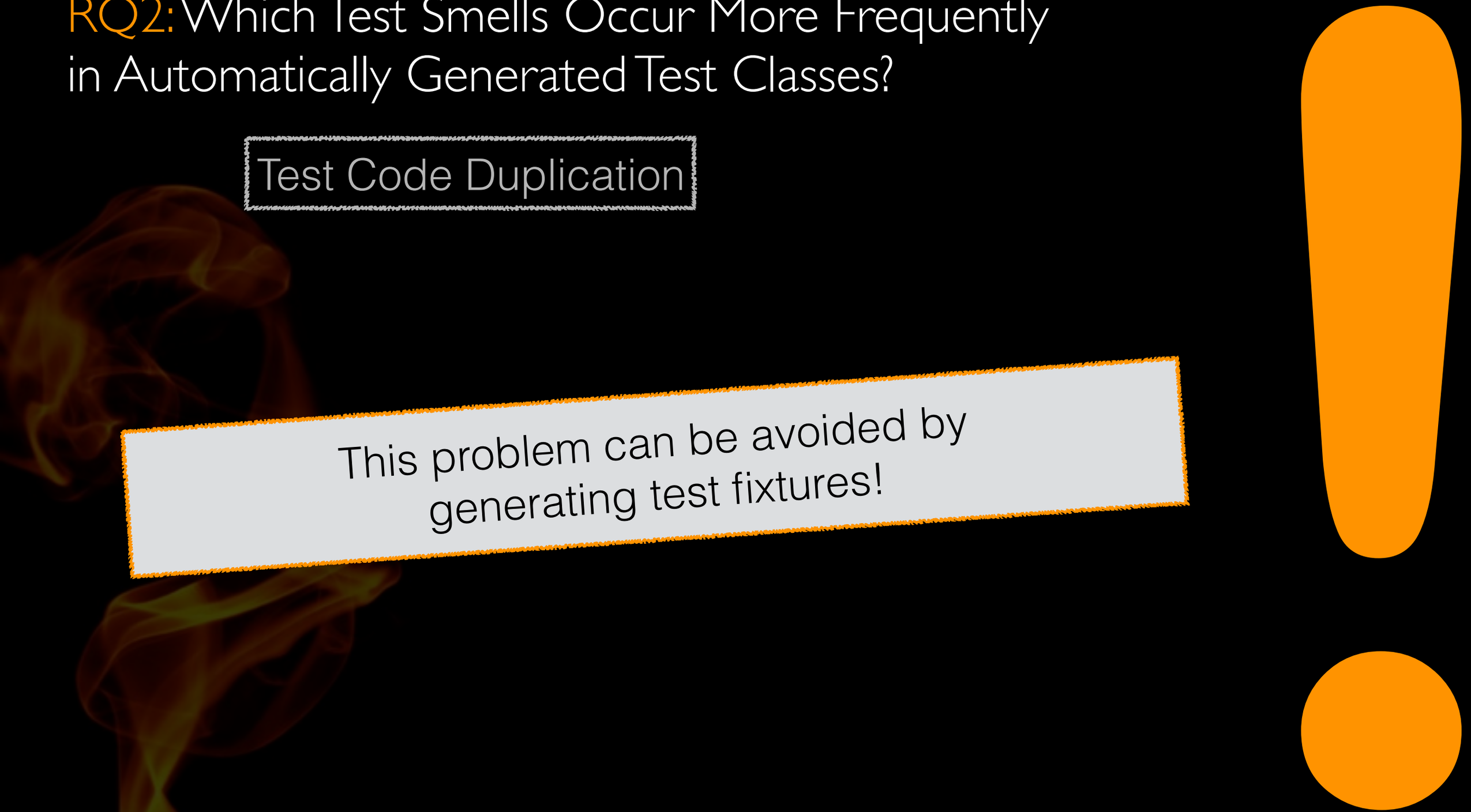
```
public void test9 () throws Throwable {
GenericProperties generic0 = new GenericProperties();
boolean boolean0 = generic0.isValidClassname();
…
}
```



Results of the Study

RQ2: Which Test Smells Occur More Frequently in Automatically Generated Test Classes?

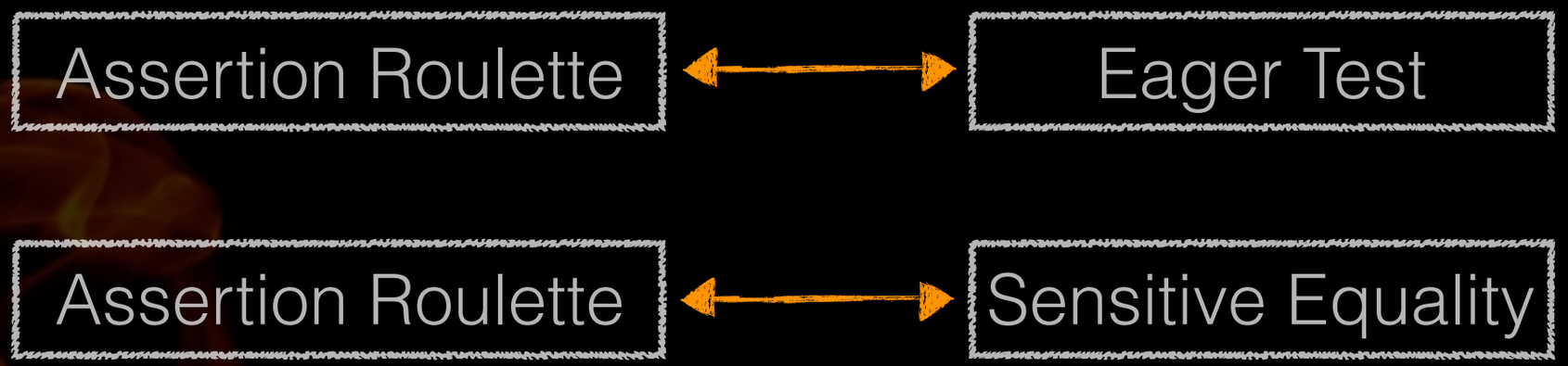
Test Code Duplication



This problem can be avoided by generating test fixtures!

Results of the Study

RQ3: Which Test Smells Co-Occur Together?



Results of the Study

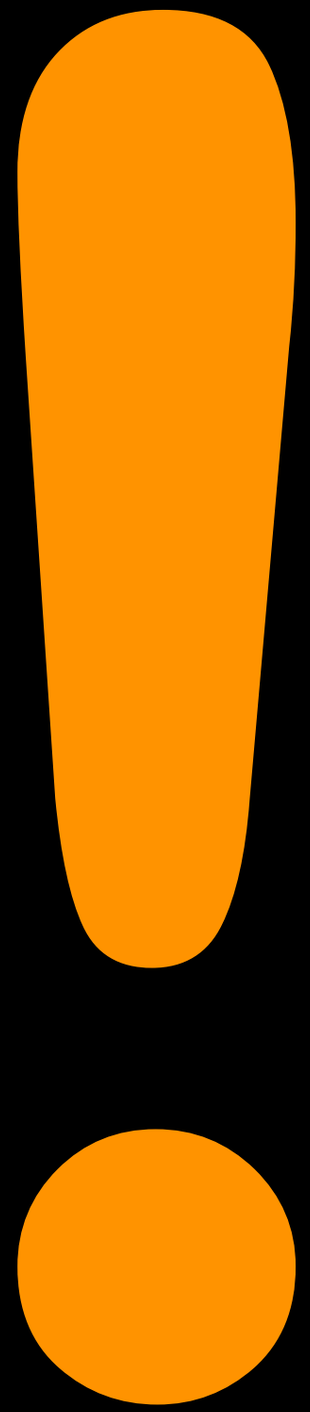
RQ3: Which Test Smells Co-Occur Together?

Assertion Roulette ⟷ Eager Test

Assertion Roulette ⟷ Sensitive Equality

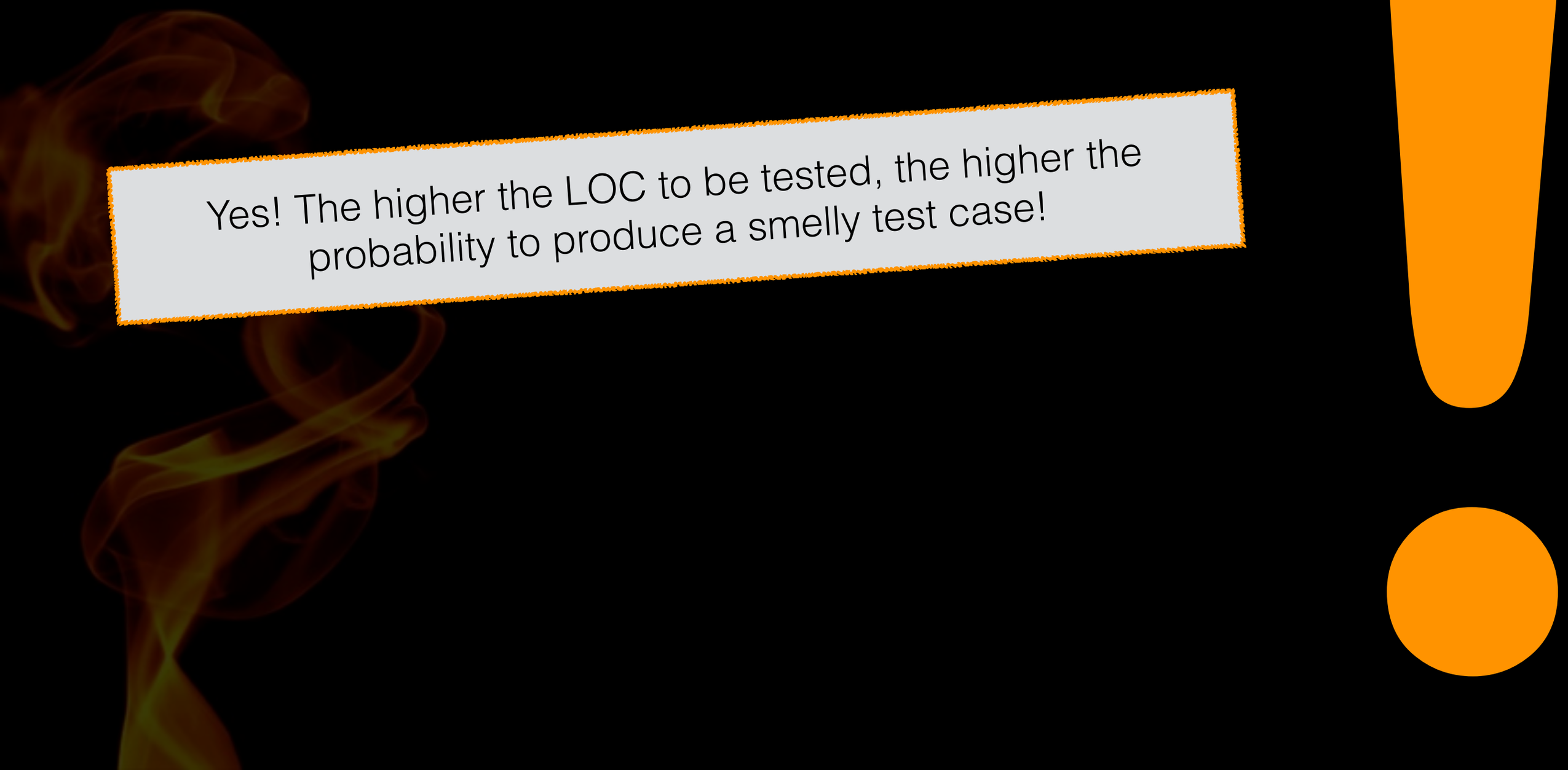
Resource Optimism ⟷ Mystery Guest

Automatic tools have as main goal that of maximize coverage, without considering test code quality



Results of the Study

RQ4: Is There a Relationship Between the Presence of Test Smells and the Project Characteristics?



Results of the Study

RQ4: Is There a Relationship Between the Presence of Test Smells and the Project Characteristics?

Yes! The higher the LOC to be tested, the higher the probability to produce a smelly test case!

The higher the LOC of the JUnit class, the higher the probability to produce a smelly test case!

Summarizing

Lesson 1: Current implementations of search-based algorithms for automatic test case generation do not consider code quality, increasing the probability to introduce smells!

Summarizing

Lesson 1: Current implementations of search-based algorithms for automatic test case generation do not consider code quality, increasing the probability to introduce smells!

NB: Considering test code quality is important not only to avoid the introduction of smells, but also because the coverage can be increased!

Automatic Test Case Generation: What If Test Code Quality Matters?
[F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, A. De Lucia - ISSTA'16]

Summarizing

Lesson 2: Automatic test case generation tools do not produce text fixtures during their computation, and this implies the introduction of several code clones in the resulting JUnit classes.

Future research should spend effort in the automatic generation of test fixtures!

From now on…

Challenge 1: Evaluating Test Smells in Test Cases automatically generated by other tools

From now on…

Challenge 1: Evaluating Test Smells in Test Cases automatically generated by other tools

Challenge 2: Defining new algorithms able to solve the design problems analyzed (e.g., test fixtures).

On the Diffusion of Test Smells in Automatically Generated Test Code: An Empirical Study

Fabio Palomba[*], Dario Di Nucci[*], Annibale Panichella[+], Rocco Oliveto[°], Andrea De Lucia[*]

SBST 2016
May, 16th 2016
Austin, TX, USA

[*] University of Salerno, Italy
[+] Delft University of Technology, The Netherlands
[°] University of Molise, Italy